USB Device Drivers: A Stepping Stone into your Kernel

Moritz Jodeit
n.runs AG
Nassauer Str. 60, D-61440 Oberursel
moritz.jodeit@nruns.com

Martin Johns
SAP Research
Vincenz-Priessnitz-Str. 1, D-76131 Karlsruhe
martin.johns@sap.com

***Abstract*—The widely–used Universal Serial Bus (USB) exposes a physical attack vector which has received comparatively little attention in the past. While most research on device driver vulnerabilities concentrated on wireless protocols, we show that USB device drivers provide the same potential for vulnerabilities but offer a larger attack surface resulting from the universal nature of the USB protocol. To demonstrate the effectiveness of fuzzing USB device drivers, we present our prototypical implementation of a mutation–based, man-in-the-middle USB fuzzing framework based on an emulated environment. We practically applied our framework to fuzz the communication between an Apple iPod device and a Windows XP system. This way, we found several potential vulnerabilities. This supports our claim that the USB architecture exposes real attack vectors and should be considered when assessing the physical security of computer systems in the future.**

I. Introduction

The Universal Serial Bus (USB) is a widely–used serial cable bus for connecting various peripherals to a host computer. Because of the widespread use and the ubiquitous nature of USB it provides an compelling attack surface. In this paper we are concentrating on attacks against device drivers and the USB stack itself.

The fact that device drivers provide the potential for exploitable vulnerabilities was already shown in [6] and [10]. But compared to 802.11 wireless device drivers, USB has the property of being a universal transport medium for further protocols. Hence, potential attacks are not limited to the USB related code inside the kernel but extend over a large number of different kernel sub-systems and device drivers reachable by USB devices which would not be associated with USB at a first glance. The USB protocol allows to reach those parts of the kernel which could otherwise not easily be attacked remotely.

This paper proposes a mutation–based USB fuzzing framework. Our approach is based on fuzzing in an emulated environment inspired by the work of Keil and Kolbitsch [9] for 802.11 wireless fuzzing. Instead of emulating USB devices in software we are attaching physically connected USB devices to the guest operating system running inside a virtual machine and fuzz the communication between the physical device and the virtual host.

Relying on a mutation–based approach gives us the flexibility to fuzz test a broad range of different device drivers without the need to emulate every single device which would be very time consuming. Doing the fuzzing in an emulated environment comes with various advantages. Besides the good target monitoring capabilities, virtual machine snapshots allow us to do exact matching between a specific USB device attachment and a potential crash.

II. Technical Background

The USB architecture can be divided into three separate parts. These are the USB devices, the USB host and the USB interconnect, which connects all USB devices with a single USB host.

USB devices are either *hubs* or *functions*. A USB hub is a special device that provides one or more attachment points to the bus, while a function provides a specific capability. Examples are a USB mouse device which usually provides a HID (Human Device Interface) function while an external hard disk drive provides a mass storage function. Each USB host controller provides a *root hub*, which is the attachment point for all connected devices.

The USB host is the central point in the USB architecture. It interacts through the host controller with the rest of the USB system. Only a single USB host per bus is allowed. Tasks of the host include the management of all transfers, detection of device attachment and removal and configuration of new devices. It is important to note, that the host plays the active part in the whole communication. All transfers are initiated by the host and USB devices only answer to requests send by the host[1].

Figure 1 shows the logical connection between a USB device and the host. Communication takes place using so called *pipes*. Pipes are unidirectional or bidirection communication channels between the host and a USB device. The end of each pipe connects to an *endpoint*.

Endpoints are compareable with IP sockets. They are the source or sink of a communication flow on the bus. Each endpoint has an associated direction which is either IN or OUT. IN endpoints transfer data from the device to the host, while OUT endpoints transfer data from the host to the device. Each USB device provides at least the endpoint 0 which is connected with the *default control pipe*. The main purpose of the default control pipe is to configure the device

[1]One exception is the USB OTG supplement[15] to the USB specification[7].

once it is attached. Depending on the purpose of the device multiple other endpoints may be provided.

Multiple pipes can be grouped into *interfaces* where each interface provides a specific functionality and is handled by a single USB device driver on the host. One interface may provide a mass storage device while a second interface may provide a USB printer.

Configurations group multiple interfaces and are mainly used to provide the same functionality with different settings. Only a single configuration can be active at a time.

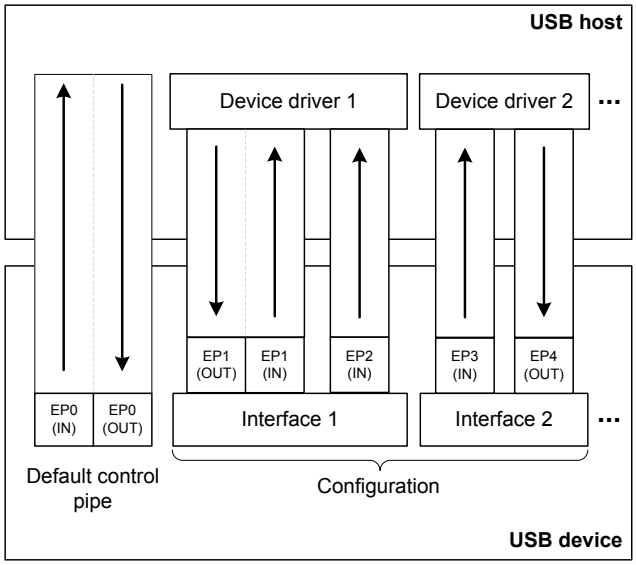

Figure 1. Logical connection between a USB device and a host

A. Device Enumeration

When a new USB device is connected to the bus through a hub the *device enumeration* process starts. Each hub provides an IN endpoint, which is used to inform the host about newly attached devices. The host continually polls on this endpoint to receive device attachment and removal events from the hub.

Once a new device was attached and the hub notified the host about this event, the USB bus driver of the host enables the attached device and starts requesting information from the device. This is done with standard USB requests which are sent through the default control pipe to endpoint zero of the device. Information is requested in terms of *descriptors*. USB descriptors are data structures that are provided by devices to describe all of their attributes. This includes e.g. the product/vendor ID, any device class affiliation, and strings describing the product and vendor. Additionally information about all available endpoints is provided.

After the host read all the necessary information from the device it tries to find a matching device driver. The details of this process are dependant on the used operating system. For lack of space we are describing the process for Microsoft Windows only but similar concepts apply to other operating systems.

B. Device Driver Loading

After the first descriptors were read from the attached USB device, the host uses the vendor and product ID from the device descriptor to find a matching device driver. Windows first tries to find the product/vendor ID combination in the registry. If the device was successfully enumerated in the past, a match is found in the registry which indicates the associated device driver to be loaded. When no match in the registry is found, Windows does a lookup in its database of available device drivers which consists of a set of INF files. Each INF file describes a set of devices for which device drivers are available.

If neither the registry nor the INF files resulted in a match for the product/vendor ID combination, Windows tries to find a matching USB class driver. Class drivers are not specific to a single device but can handle a broad range of different devices which behave according to a class specification. Most operating systems provide a set of USB class drivers which allows some classes of USB devices to be connected without the need to install a separate device driver. The latest version of Windows comes with class drivers for many of the defined USB classes [14], such as the mass storage, audio or printer class.

To find a matching class driver, Windows uses the class, subclass and protocol values read from the descriptors. The same lookups as detailed above for the product/vendor ID combination are performed.

After a matching device driver was found and loaded, it's the task of the device driver to select one of the provided device configurations. The device driver selects one of the configurations based on its own capabilities and the available bandwidth on the bus and activates this configuration on the attached device. At this point, all interfaces and their endpoints of the selected configuration are set up and the device is ready for use.

III. Attack Scenarios

In the case of the USB 2.0 standard [7], an attacker needs physical access to a system. Although nearly every system can be broken into with enough physical access, USB ports represent a special case. Often the system itself together with human interface devices, such as keyboards and mice, is protected against unauthorized access. However, USB ports are often considered safe to be provided to the user. In some cases, USB ports must even be provided to the user to accomplish the task of the respective system. USB-based hardware security tokens are one example.

If the attacker is an employee of a company he is trying to attack, he has lots of possibilities to unobtrusively

attach malicious USB devices. But even if the attacker isn't associated with the company to be attacked, there are lots of cases, where the attacker himself doesn't need direct physical access but can get his malicious USB device attached to the USB port of a system by other means.

People with legitimate physical access to a system could be paid or bribed to act in the interest of the attacker. An example could be any employee or facility staff member that might have a financial interest.

Instead of bribery, people with legitimate physical access could also be tricked to attach an attacker–supplied device. When it comes to physical access, social engineering works very well. An attacker can either just place a few attractive or interesting looking USB devices in front of a company or just send them directly by mail to the victim. Depending on how much money the attacker has available for the attack, the USB device can be in original package and could have diverse appearances, ranging from a simple USB flash drive up to an exclusive mobile phone with USB connectivity.

Another example where an attacker could trick other people to attach a malicious USB device to a system of interest is digital voting systems using so–called *digital voting pens* [2]. This is a system to speed up vote counting where each voter does his votes using a digital pen which records the coordinates of the vote using a small camera inside the pen. After the voter finished voting, the pen is given back to the election supervisor, who in turn attaches the pen to a USB docking station that is connected to the computer system used to store all votes. An attacker could either replace or modify the voting pen given to him, which would then get attached to the host system storing all the votes. A successful attack might then be used for election fraud.

Finally, the requirement of physical access might change with the Certified Wireless USB (CWUSB) extension [1] that introduces wireless USB.

IV. Attack Vectors

An enabled USB port provides various attack vectors for a connected device. Potential attacks can go far beyond USB stack and device driver attacks. Figure 2 gives a simplified overview of the different components of a typical USB host architecture. At the bottom, we have the electrical layer. Its purpose is to encode and decode the electrical signals on the wire. The electrical layer connects directly to the USB stack, which is responsible for handling protocol details of the USB protocol. Each device driver registers itself at the USB stack. The only way a USB device driver can communicate with an attached device is through the USB stack. Consequently, the first attack target is the USB stack itself.

The name "*Universal* Serial Bus" already suggests that a wide range of different classes of devices can be connected through USB. To provide their service to an attached device,

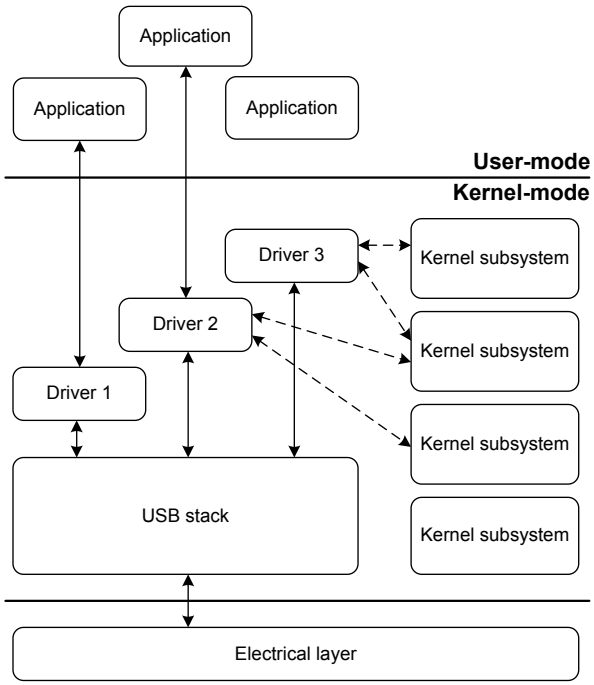

Figure 2. Relation between components of the USB host architecture

in many cases USB device drivers don't run in isolation but communicate with various other kernel subsystem components. For example, a USB network card driver makes use of the network subsystem, while a mass storage device driver utilizes the I/O and SCSI subsystem of the kernel. Even when receivers for other protocols, such as IrDA, 802.11 or Bluetooth, are disabled, a connected USB device can still pretend to be of the respective communications class and, thus, get access to the protocol stacks otherwise not reachable for external attacks.

Finally, USB devices are not exclusively connected to kernel subsystems. Applications running in user–mode can communicate with USB devices, e.g., to provide the interaction with a user. Hence, data coming from a malicious USB device can reach applications running in user–mode which increases the attack surface even further.

V. Implementation

To practically find potential vulnerabilities in the components listed in the previous section, we built a USB fuzzing framework. This section discusses our design decisions and implementation.

A. System Design

The first consideration when building a USB fuzzer is the decision between a generation–based and mutation–based fuzzer [13]. The effort to build a complete generation–based fuzzer is comparable with the development of a new USB device driver and, thus, could get very time–consuming.

Sticking to a mutation–based fuzzer releases us from the task of emulating a USB device to get a specific device driver loaded and fuzzed. We just attach the corresponding device and modify the USB packets in transit. Consequently, a mutation–based fuzzer is the preferred choice for quickly getting first results.

To implement a mutation–based fuzzer we need a way to intercept the communication between an attached device and the USB host. The first option is to do the fuzzing on the target host itself. A small kernel component could be developed which would intercept the USB packets just before they are delivered to the respective device driver to be tested. Although this may be quickly implemented it has the disavantage that it is platform-specific. The other problem is that the fuzzing happens on the host we are trying to crash.

With the requirement that fuzzing should happen before the USB packets reach the target host, there are two possibilities. The first option is to utilize a hardware–based approach which enables us to physicaly connect the USB fuzzer to the target host. This would allow us to fuzz–test any device as long as a USB port is provided. The disadvantage is that it requires special–purpose hardware.

To overcome this limitation we chose the second option and perform the fuzzing in an emulated environment inspired by the work of Keil and Kolbitsch [9] for 802.11 wireless fuzzing. The use of an emulated environment allows us to do the fuzzing before the USB packets reach the host but still gives us the freedom to build a software–only solution. Additionally, we get all the benefits of fuzzing in an emulated environment.

Besides the good automation and target monitoring capabilities of emulated environments one of the most useful features for our task are virtual machine snapshots. These allow us to store a snapshot of the current CPU, memory and disk state which can be restored at a later point in time. When fuzzing USB device drivers some systems were found to disable the USB port after repeatedly attaching malformed USB devices. To continue fuzzing in such a case, a reboot of the target system would normally be required. Virtual machine snapshots allow us to just restore to a known good state and continue fuzzing.

Another problem with repeatedly attaching fuzzed USB devices is the fact, that some memory corruption may not result in an immediate crash of the target host. A crash could happen at a later point in time triggered by some unrelated event. This complicates the linking of encountered crashes to a specific device attachment. To link each crash to one specific attachment, virtual machine snapshots can be used to restore a known good state after each attachment.

B. Architecture

To modify the USB communication between a USB device and the host we propose the man–in–the–middle architecture shown in Fig. 3. It's based on three main components:

1) Receiving Component
2) Processing Component
3) Device Emulation Component

The receiving component is responsible for acquiring the USB packets from an attached USB device. It either talks directly to the connected device or reads in a stored flow of communication, which was recorded beforehand. All USB packets are just forwarded between the USB device and the processing component.

The processing component conducts the optional modification or analysis of the USB communication. This is where the actual fuzzing or analysis of the raw USB packets can be implemented. The processing component can also record a flow of communication and store it for replaying at a later point in time. The processing component passes all the USB communication between the receiving component and the device emulation component.

The device emulation component forwards the USB communication it received from the processing component to a connected host system. From the perspective of the host, it acts like the real USB device.

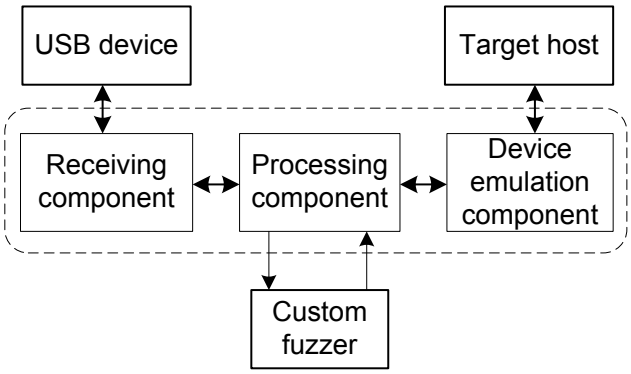

Figure 3. Man–in–the–middle architecture

Our implementation is based on the QEMU machine emulator [4]. QEMU can emulate a complete PCI UHCI USB controller. Besides USB devices which are emulated directly by QEMU, it also allows to pass–through physically connected USB devices to the guest operating system. We utilize this functionality and implement the receiving component and the device emulation component directly into QEMU as a set of patches. The final architecture is shown in Fig. 4.

The receiving component passes on all USB packets between a physical USB device and the processing component. To get access to the physical USB device our implementation makes use of QEMU which in turn uses the *USB device file system*. This is a Linux file system that provides all the needed hardware details of attached

USB devices to user–mode applications. To retrieve the descriptors of an attached USB device, the corresponding device files inside the mounted USB device file system can be read. Communication with a device takes place using `ioctl()` calls on the desired device file.

The dependance on QEMU for the receiving component instead of using the USB device file system directly is basically due to the fact that our current implementation is heavily based around QEMU. To fully take advantage of the modular design, future versions will make use of the USB device file system directly.

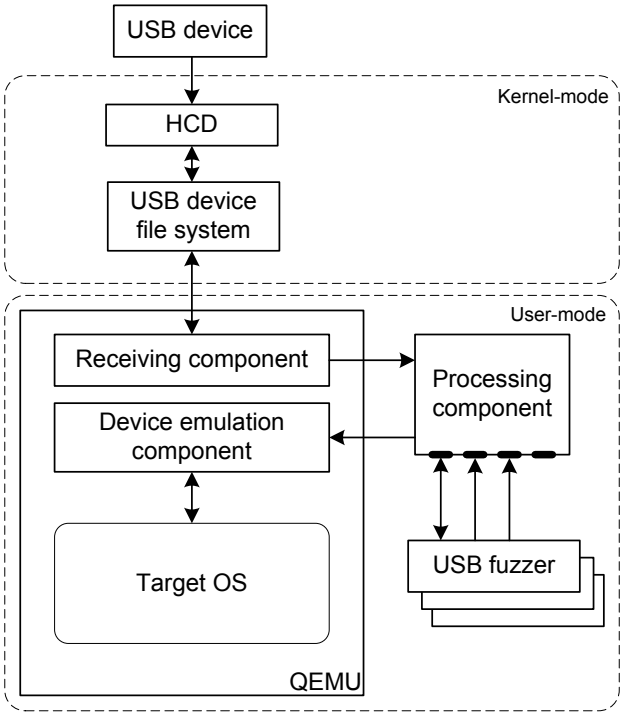

Figure 4. Design of the mutation–based fuzzing framework

The processing component is implemented externally as a Python library which is connected to the receiving component and the device emulation component using a set of named pipes. All USB packets exchanged between a USB device and the host are passed through the processing component. The processing component doesn't implement any functionality itself but just provides simple Python bindings for easy access to the raw USB packets. Those bindings can be utilized by third–party extensions to easily create custom fuzzers or analysis tools.

VI. Evaluation

To evaluate our implementation we created a simple fuzzer based on our framework which just randomly replaces bytes in the USB packets exchanged between the device and host. All IN packets are randomly selected for fuzzing while OUT packets are ignored. For each selected packet, a random number of bytes of the packet are replaced with random values while the most significant bit of each byte is set more frequently in the hope to trigger signedness issues.

As a fuzzing target we chose an Apple iPod Shuffle device connected to a host system running Windows XP SP2 without additional patches. The only software additionally installed was the latest release of Apple's iTunes software[2]. The iPod Shuffle identifies itself as a mass storage device and, thus, is handled by the USB mass storage class driver of Windows XP. The reason we chose this device instead of some usual USB flash drive is because of the massive communication taking place just after it is attached. After the mass storage device is detected by the system the iTunes software is launched by a service running in the background which is installed as part of the iTunes application. The loaded iTunes application then reads various information from the device's file system leading to multiple USB packets being exchanged. Fuzzing those packets gives us a good chance to reach various kernel components as well as the iTunes service responsible for the detection of attached iPods and the iTunes application running in user–mode.

The fuzz test consisted of repeatedly attaching the device, letting the host talk to the device for some time and then detaching it again. While doing this the state of the host was monitored to detect any anomalies or crashes. All these actions were performed using the Python API provided by our fuzzing framework.

During the tests multiple bug checks were triggered leading to a kernel crash. The crashes encountered were triggered at various locations. While two of the kernel crashes happened inside the USB host controller driver, one crash was triggered in the USB mass storage driver and another one was triggered in the file system code responsible for reading the partition table from the attached device. Additionally a crash in the user–mode iPod service binary was triggered.

Although no deep analysis of the found crashes was performed, at least one of them was caused by memory corruption, making it probably exploitable. See Table I for the complete list of found crashes.

Component	Result
usbuhci.sys	Kernel crash (Bug check 0xfe)
usbuhci.sys	Kernel crash (Bug check 0xbe)
usbstor.sys	Kernel crash (Bug check 0xc2)
disk.sys	Kernel crash (Bug check 0x7e)
iPodService.exe	Application crash

Table I
Apple iPod fuzzing results

[2]Apple iTunes 8.1.1.10 was used.

VII. Related Work

The dominant focus of attention in respect to exploiting memory corruption vulnerabilities in device drivers has been on the realm of wireless communication. Several publications, such as [6] or [10], detail the complexity of 802.11 wireless device drivers and the resulting potential for vulnerabilities. Much effort has been put into the development of IEEE 802.11 wireless fuzzers [5] to practically find those vulnerabilities. While most publications concentrate on device drivers for wireless protocols, Barrall and Dewey showed in [3] that USB stacks and device drivers also provide the potential for vulnerabilities. They demonstrate their point with a vulnerability in USB related code of the Windows operating system. No details were made public though. Rafael Dominguez Vega continued research in that direction and demonstrated the exploitation of a Linux USB device driver bug in [17] using a custom-built USB device. Details about the actual vulnerability being exploited were not disclosed. He also described some first USB fuzzing techniques.

The idea to use an emulated environment for fuzzing 802.11 wireless device drivers was first demonstrated by Keil and Kolbitsch in [9]. They utilize the emulated environment to circumvent the hard timing constraints when fuzzing 802.11 device drivers. The implementation of our USB fuzzing framework is based on this idea.

Furthermore, first approaches towards exploiting drivers drivers outside the wireless realm have been made: Ilja van Sprundel showed in [16] how to utilize fuzzing to uncover vulnerabilities in filesystem drivers. As the USB protocol grants a malicious device direct access to the system's filesystem, van Sprundel's work is of high relevance in the context of this paper.

Finally, instead of exploiting implementation flaws in device drivers, Maximillian Dornseif demonstrated in [8] how the use of DMA in FireWire empower an attacker to read and write arbitrary physical memory of the host. Dornseif's work work was refined by Piegdon and Pimenidis in [12] towards arbitrary code execution. Such attacks require the device to be the controlling instance on the bus which is in general not the case with USB. In the USB protocol the host controls all communication on the bus. However, David Maynor showed in [11] that DMA attacks against USB are nevertheless possible by utilizing the USB OTG extension [15] which allows USB devices to provide limited USB host functionality to communicate directly with other USB devices which would not be possible otherwise.

VIII. Conclusion and Future Work

In this paper we discussed security implications of the Universal Serial Bus. After raising the awareness by listing potential attack scenarios, we explored the large attack surface provided by enabled USB ports. Subsequently, we described our implementation of a mutation–based USB fuzzing framework utilizing an emulated environment. By using a fuzzer together with an iPod Shuffle device we demonstrated that USB device drivers not only provide the potential for vulnerabilities but can also be used as a stepping stone to trigger vulnerabilities in other kernel components not directly related to USB and even in user–mode applications communicating with attached devices. To find those vulnerabilities we used a simple random–based fuzzer without any knowledge of the underlying protocols being fuzzed. It is to be expected that the development of more intelligent fuzzers will result in even better results. We paved the way for the development of such fuzzers with our fuzzing framework.

Although we exclusively focused on attacks against the USB host in this paper, the presented framework can also be applied in reverse direction to fuzz test physical USB devices, such as smartphones or PDAs. This might provide a potential area for future research.

Despite the fact that fuzzing in an emulated environment provides various benefits, the actual exploitation of a vulnerability in a real–world scenario requires the use of hardware which emulates a physical USB device. Further research into the creation of a separate hardware–based device emulation component is required.

Another area of interest for future research is the Certified Wireless USB (CWUSB) extension [1]. One of the design goals of the wireless USB specification is to keep the current software infrastructure including all the USB device drivers intact. Wireless USB provides a wireless transport mechanism for the USB protocol and, thus, makes attacks against USB even more interesting since physical access is no longer required. In this context, CWUSB's mechanisms used for authentication and encryption should be analyzed for their effectiveness.

References

[1] Agere, Hewlett-Packard, Intel, Microsoft, NEC, Philips, and Samsung. *Wireless Universal Serial Bus Specification 1.0*, May 2005.

[2] Joerg Arzt-Mergemeier, Willi Beiss, and Thomas Steffens. The Digital Voting Pen at the Hamburg Elections 2008: Electronic Voting Closest to Conventional Voting. In *E-Voting and Identity*, volume 4896/2007 of *LNCS*. Springer, 2007.

[3] Darrin Barrall and David Dewey. "Plug and Root," the USB Key to the Kingdom. Presentation at Black Hat USA, July 2005.

[4] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005*, page 41. USENIX Association, 2005.

[5] Laurent Butti. Wi-Fi Advanced Fuzzing. Presentation at Black Hat Europe, 2007.

[6] Johnny Cache and David Maynor. Device Drivers: Don't build a house on a shaky foundation. Presentation at Black Hat USA, August 2006.

[7] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. *Universal Serial Bus Specification 2.0*, 2000.

[8] Maximillian Dornseif. 0wned by an iPod. Presentation at PacSec, November 2004.

[9] Sylvester Keil and Clemens Kolbitsch. Stateful Fuzzing of Wireless Device Drivers in an Emulated Environment. White Paper, Secure Systems Lab, http://www.iseclab.org/papers/fuzz_qemu.pdf (2009/05/17), September 2007.

[10] David Maynor. Device Drivers 2.0. Presentation at Black Hat DC, February 2007.

[11] David Maynor. 0wn3d by everything else: USB/PCMCIA Issues. Presentation at CanSecWest, May 2005.

[12] David R. Piegdon and Lexi Pimenidis. Targeting Physically Addressable Memory. In Robin Sommer Bernhard M. Haemmerli, editor, *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2007)*, volume 4579 of *LNCS*. Springer, July 2007.

[13] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[14] USB Implementers Forum, Inc. Approved Class Specification Documents. http://www.usb.org/developers/devclass_docs (2008/05/22).

[15] USB Implementers Forum, Inc. On-The-Go Supplement to the USB 2.0 Specification, 2006.

[16] Ilja van Sprundel. Fuzzing: Breaking Software in an Automated Fashion. talk at the 22th Chaos Communication Congress (22C3), http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf, December 2005.

[17] Rafael Dominguez Vega. "USB Attacks: Fun with Plug and 0wn. Presentation at Defcon 17, August 2009.